# **领域驱动架构（DDD）建模中的模型到底是什么？**

要回答这个问题，需要三步走：为什么要建模；怎么建模才合理；“领域”模型具体指什么。

## **为什么要建模**

客户在专卖店买了个手机，留下了自己的名字和电话，店员做了记录。客人来时，只要店员能在记录里查到客人名字和电话的订单，就说明客人曾经买过手机。

什么人需要查看订单呢？店员 A 需要查看，店员 B 也需要查看。客人来咨询的时候，应该能随时调取。老板也需要查看，用来汇总销售情况。大家都要看，格式就必须统一，要不然有的只记了电话，有的只记了名字，有的什么都没记，就乱套了。

大家商量之后决定：订单必须包括客户名字、电话和购买的商品。那么就有“订单 = 名字 + 电话 + 商品信息”。这是店员和老板的**心智模型**（mental model）。

要用一个数字系统来支持订单的管理，必须形成对应的**数据模型**（data model），称之为**数据建模**（data modeling），中文简称**建模**。电脑采用数字化的精确储存，所以数据的格式必须提前明确，比如名字是 2-4 个中文字符，电话是 11 位数字等等。

建模本质上是一种抽象。抽象就是归类，其目的是减轻认知的负担，避免重复的思考和工作，提升人的计算能力。所以，“通用”是建模的第一步，接下来我们还需要“复用”建好的模型。

假设手机卖出之后，客户需要维修服务。 客户来到店里，询问店员，店员查询确认了订单，然后把客人引到门店旁边的维修中心。维修中心的工程师拿到订单，发现手机已经过了保修期，所以他写了一个维修单，把客户的名字、电话、手机信息、维修费用写到上面。客户交了费，拿到修好的手机，走了。

这引出一个问题。维修中心需要的客户信息，其实在店员那边有，没有必要自己再抄一遍，否则很容易出错，还会遇到信息同步的问题。那么，我们就需要再做一次建模，把客户的名字和电话从订单模型中拿出来，单独做一个客户模型。订单和维修单都复用这个客户模型。

这样，我们就得到了如下三个模型。

* 订单 = 客户 + 商品信息
* 维修单 = 客户 + 商品信息 + 维修信息
* 客户 = 名字 + 电话

这里，维修单模型里面似乎包含了一个完整的订单（客户 + 商品信息），为什么不直接复用订单呢？也许是因为维修部门也负责别的地方购买的商品。另外，客户的名字和电话更新之后，是不是要直接修改已经完成的订单和维修单呢？值得商榷，已经完成的信息不应该有意料之外的变化。还有，如果客户是一个单独的模型，那么背后的团队会是怎么样呢？需要仔细考虑。

当数据模型可以完全覆盖业务需要，建模也就初步完成了。

那么，为什么要建模？第一，要把心智模型提取出来，显性化，让不同的人对业务的理解达成一致；第二，要归类复用，避免重复的工作，让人可以关注更高层面的事务。

如前面所示，即便是建立简单的模型，我们也需要诸多考虑。把这些需要考虑的点体系化，就引出了下一个问题：怎么建模才合理？

## **怎么建模才合理**

判断模型好坏的重要依据是它的使用效率（扩展度、灵活度、与组织的对应度），所以建模的合理性也围绕这个来展开。

为了形成高扩展度、高灵活度并且顺应康威定律的模型，Eric Evans 汇总并命名了 DDD 这种方法论。它的独特之处，是它认可了“人”这种生物在做抽象过程中的一些必然缺憾，并且提出了一些解决方案。

首先，它明确指出了“自然语言”这个工具的不精确性。其次，它明确指出了人在设计和实现上的矛盾性，即：人类的预测能力很差，所以我们拥抱变化；但同时，我们不能纯寄希望于误打误撞地演化出一个好结果，应该提前深思熟虑地做好设计，再辅之以小的改进。

语言的不精确性，可以用“普通话”（Ubiquitous Language）来解决，即大家在讨论任何东西之前先规范语言，统一词汇的定义。兼顾稳固和灵活的设计，则由良好的分层来做到。

DDD 把模型分成四层。

* UI 层，负责界面展示。
* 应用层（Application Layer），负责业务流程。
* 领域层（Domain Layer），负责领域逻辑。
* 基建层（Infrastructure Layer），负责提供基建。

分类的依据是：越往上，预期变动越频繁；越往下，预期变动越少。

对于程序员来说，UI 和基建应该很容易分清，一个只管展示，一个只管提供持续储存、网络传输等等基础设施，都没有“业务”的参与。容易混淆的是应用层和领域层，在这两层中存在的，就是应用模型和领域模型。

这就引出了下一个问题：到底什么是领域模型？

## **什么是“领域”模型**

按 DDD 的定义，领域模型应该捕捉“业务规则”或者“领域逻辑”（business rules / domain logic），而应用模型则捕捉“应用逻辑”（application logic）。

模型属于哪一层，有个粗略的判断方式。如果是一个实体（entity）和针对实体的增删改查，就属于领域层；如果是一个场景， 比如出现在 UI 菜单上的选项，就属于应用层。

比如：账单，用户，编辑商品，编辑库存，这些是领域层；“购买商品”，则是应用层。

前者是针对实体的操作。每一个实体都只有增删改查这样的操作。与之相反的是，要完整实现“购买商品”这个场景，也许需要检查库存、创建订单、创建交易等多个操作。

这样看来，领域模型就像是数据库的表。不过，除了字段定义之外，领域模型还需要有领域逻辑。关系型数据库通常可以部分实现领域逻辑，比如使用外键、自增 ID 等等，但是更为复杂的领域逻辑则需要用代码来实现。比如，在创建订单的时候，需要扣除相应数量的库存；当订单失效，则需要恢复库存。

我们可以从两个方面理解领域逻辑。

第一，领域逻辑就是显性的专业知识，是相对容易理解和学习的部分。买了东西要给钱，出了货要扣库存，飞机来了要腾跑道，炮弹来了要拦截，这些都是专业知识中符合逻辑，可以很容易地推导和学习。与之相对的是隐性的专业知识，或者 Martin Fowler 所说的“业务非逻辑”（business illogic），指的是逻辑很难推论的软知识，与人相关的部分，与不可抗拒的、稀奇古怪的意外情况相关的部分。

第二，领域逻辑是提纯、通用的规则。不管是买手机、买房、买汽车，都会有创建订单、创建交易这样的零售领域模型，它是高度提纯、通用的。如果我们需要去修改领域模型，说明我们已经进入了另一个领域。此外，规则的存在是为了维护某个事物，这个事物，就是领域模型的正确、完整性。对领域模型的正常操作，总会给我们一个“合规”的领域。

不管我们怎么去玩弄订单、交易、商品模型，增删改查，乱搞一通，最后出来的结果都应该符合规则。创建订单的时候，订单模型会尝试减库存，成功，则创建，失败，则不允许创建，如此云云。最后出来的结果不会有逻辑问题，比如订单上的商品不存在，或者交易对应的订单不存在。

当然，领域模型只管“合规”，但不管“合理”。大型超市里，一位收银员决定把所有货品下架，这是否合理？这个不属于领域模型关心的范围，领域模型只知道，要把货物库存的位置从货架转移到仓库，目标仓库必须有能存放这个货物的空位，转移完毕时从货架的可用空间中减去货物的大小。至于谁做，为什么做，能不能做，这么做合不合理，（通常）不在领域层的关心范围，而是放到应用层去做。

在前面的例子中，如果一个客户在黑名单中，不允许购买，那么这个检查，通常是在应用层去检查。这样，我们就可以很容易地复用领域模型或者调整应用规则，而不至于把易变的应用规则混到稳定的业务规则里去。

## **模型的设计和实现**

那么，DDD 的模型属于业务描述还是代码、数据库定义？作者想表达的意思很明确，二者都是。好的业务描述应该能非常好地对应到代码，代码也应该以最清晰地方式来呈现业务描述。当然，转换和适配肯定不可少，毕竟有先例摆在那。想要达到设计即代码的 UML，尸体都还是温的。

设计和实现的最佳契合点，其实就在“界面”之中。“界面”这个名字本身已经体现了这层意思。所以，建好的模型，和暴露出来的界面、接口，应该有相当高（或者完全一致）的对应关系。这个界面，可能是一个类的公开成员函数，也可能是一个微服务上暴露出来的 RESTful 接口，或者是一个公开的普通 Web API，都没有差，只要和设计符合就行。

最后，总结起来，就是：

* DDD 把业务分成 UI、应用、领域、基建四层，其核心是高度提纯、通用、少变化的领域层，是谓“领域驱动”；
* 领域层中包含领域模型，捕捉领域逻辑，暴露出接口用于操作领域模型，这些接口提供的操作可以确保领域是自洽的；

领域模型既是业务描述，又是代码实现的结构设计，二者的结合点在于公开出来的界面、接口。